2 Address Obfuscation

2.1 Obfuscating Transformations

The objectives of address obfuscation are to (a) randomize the absolute locations of all code and data, and (b) randomize the relative distances between different data items. These objectives can be achieved using a combination of the following transformations:

I. Randomize the base addresses of memory regions.

By changing the base addresses of code and data segments by a random amount, we can alter the absolute locations of data resident in each segment. If the randomization is over a large range, say, between 1 and 100 million, the virtual addresses of code and data objects become highly unpredictable. Note that this does not increase the physical memory requirements; the only cost is that some of the virtual address space becomes unusable. The details depend on the particular segment:

1. *Randomize the base address of the stack.* This transformation has the effect of randomizing all the addresses on the stack. A classical stack-smashing attack requires the return address on the stack to be set to point to the beginning of a stack-resident buffer into which the attacker has injected his/her code. This becomes very difficult when the attacker cannot predict the address of such a buffer due to randomization of stack addresses. Stack-address randomization can be implemented by subtracting a large random value from the stack pointer at the beginning of the program execution.
2. *Randomize the base address of the heap.* This transformation randomizes the absolute locations of data in the heap, and can be performed by allocating a large block of random size from the heap. It is useful against attacks where attack code is injected into the heap in the first step, and then a subsequent buffer overflow is used to modify the return address to point to this heap address. While the locations of heap-allocated data may be harder to predict in long-running programs, many server programs begin execution in response to a client connecting to them, and in this case the heap addresses can become predictable. By randomizing the base address of the heap, we can make it difficult for such attacks to succeed.
3. *Randomize the starting address of dynamically-linked libraries.* This transformation has the effect of randomizing the location of all code and static data associated with dynamic libraries. This will prevent *existing code* attacks (also called *return-into-libc* attacks), where the attack causes a control flow transfer to a location within the library that is chosen by the attacker. It will also prevent attacks where static data is corrupted by first corrupting a pointer value. Since the attacker does not know the absolute location of the data that he/she wishes to corrupt, it becomes difficult for him/her to use this strategy.
4. *Randomize the locations of routines and static data in the executable.* This transformation has the effect of randomizing the locations of all functions in the executable, as well as the static data associated with the executable. The effect is similar to that of randomizing the starting addresses of dynamic libraries.

We note that all of the above four transformations are also implemented in the PaX ASLR system, but their implementation relies on kernel patches rather than program transformations. The following two classes of transformations are new to our system. They both have the effect of randomizing the relative distance between the locations of two routines, two variables, or between a variable and a routine. This

makes it difficult to develop successful attacks that rely on adjacencies between data items or routines. In addition, it introduces additional randomization into the addresses, so that an attacker that has somehow learned the offsets of the base addresses will still have difficulty in crafting successful attacks.

II. Permute the order of variables/routines.

Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in which the variables appear. Such permutation makes it difficult to predict the distance accurately enough to selectively overwrite security-critical data without corrupting other data that may be critical for continued execution of the program. Similarly, attacks that exploit relative distances between code fragments, such as partial pointer overflow attacks (see Section 3.2.3), can be rendered difficult by permuting the order of routines. There are three possible rearrangement transformations:

1. *permute the order of local variables in a stack frame*
2. *permute the order of static variables*
3. *permute the order of routines in shared libraries or the routines in the executable*

III. Introduce random gaps between objects.

For some objects, it is not possible to rearrange their relative order. For instance, local variables of the caller routine have to appear at addresses higher than that of the callee. Similarly, it is not possible to rearrange the order of malloc-allocated blocks, as these requests arrive in a specific order and have to be satisfied immediately. In such cases, the locations of objects can be randomized further by introducing random gaps between objects. There are several ways to do this:

1. *Introduce random padding into stack frames.* The primary purpose of this transformation is to randomize the distances between variables stored in different stack frames, which makes it difficult to craft attacks that exploit relative distances between stack-resident data. The size of the padding should be relatively small to avoid a significant increase in memory utilization.
2. *Introduce random padding between successive* `malloc` *allocation requests.*
3. *Introduce random padding between variables in the static area.*
4. *Introduce gaps within routines, and add jump instructions to skip over these gaps.*

Our current implementation supports all the above-mentioned transformations for randomizing the base addresses of memory regions, none of the transformations to reorder variables, and the first two of the transformations to introduce random gaps.

2.2 Implementation Issues

There are two basic issues concerning the implementation of the above-mentioned transformations. The first concerns the timing of the transformations: they may be performed at compile-time, link-time, installation-time, or load-time. Generally speaking, higher performance can be obtained by performing transformations closer to compilation time. On the other hand, by delaying transformations, we avoid making changes to system tools such as compilers and linkers, which makes it easier for the approach to be accepted and used. Moreover, performing transformations at a later stage means that the transformations can be applied to proprietary software that is distributed only in binary form.

The second implementation issue is concerned with the time when the randomization amounts are

determined. Possible choices here are (a) transformation time, (b) beginning of program execution, and (c) continuously changing during execution. Clearly, choice (c) increases the difficulty of attacks, and is hence preferred from the point of security. Choices (a) or (b) may be necessitated due to performance or application binary interface compatibility considerations. For instance, it is not practical to remap code at different memory locations during program execution, so we cannot do any better than (b) for this case. In a similar manner, adequate performance is difficult to obtain if the relative locations of variables with respect to some base (such as the frame pointer for local variables) is not encoded statically in the program code. Thus, we cannot do any better than choice (a) in this case. However, choice (a) poses some special problems: it allows an attacker to gradually narrow down the possibilities with every attack attempt, since the same code with the same randomizations will be executed after a crash. To overcome this problem, our approach is to periodically re-transform the code. Such retransformation may take place in the background after each execution, or it may take place after the same code is executed several times. With either approach, there still remains one problem: a local attacker with access to such binaries can extract the random values from the binary, and use them to craft a successful attack. This can be mitigated by making such executables unreadable to ordinary users. However, Linux currently makes the memory maps of all processes to be readable (through the special file `/proc/pid/maps`), which means a local user can easily learn the beginning of each memory segment, which makes it much easier to defeat address obfuscation. In particular, the attacker can easily figure out the locations of the code segments, which makes it possible to craft existing code attacks. This is a limitation of our current implementation.

Our approach is to delay the transformation to the latest possible stage where adequate performance is obtainable. In our current implementation, the transformation is performed on object files (i.e., at link-time) and executables. For ease of implementation, we have fixed many randomizations at transformation time, such as the gaps introduced within the stack frame for any given function, the locations where libraries are loaded, etc. This means that programs have to be periodically (or frequently) re-obfuscated, which may be undesirable as the obfuscation interacts with other security procedures such as integrity-checking of executables. We therefore plan to move towards options (b) and (c) in the future.

Next, we describe our approach for implementing most of the above-mentioned transformations. Our implementation targets the Intel x86 architectures running ELF-format [*30*] executables on the Linux operating system.

2.3 Implementation Approach

Our implementation transforms programs at the binary level, inserting additional code with the LEEL binary-editing tool [*40*]. The main complication is that on most architectures, safe rewriting of machine code is not always possible. This is due to the fact that data may be intermixed with code, and there may be indirect jumps and calls. These two factors make it difficult to extract a complete control-flow graph, which is necessary in order to make sure that all code is rewritten as needed, without accidentally modifying any data. Most of our transformations, such as stack base randomization are simple, and need to be performed in just one routine, and hence are not impacted by the difficulty of extracting an accurate control-flow graph. However, stack-frame padding requires a rewrite of all the routines in the program and libraries, which becomes a challenge when some routines cannot be accurately analyzed. We take a conservative approach to overcome this problem, rewriting only those routines that can be completely analyzed. Further details can be found in Section 2.3.4.

2.3.1 Stack base address randomization

The base address of the stack is randomized by extra code which is added to the text segment of the

program. The code is spliced into the execution sequence by inserting a jump instruction at the beginning of the main routine. The new code generates a random number between 1 and 10^8, and decrements the stack pointer by this amount. In addition, the memory region corresponding to this ``gap'' is write-protected using the `mprotect` system call. The write-protection ensures that any buffer overflow attacks that overflow beyond the base of the stack into the read-only region will cause the victim program to crash.

2.3.2 DLL base address randomization

In the ELF binary format, the program header table (PHT) of an executable or a shared library consists of a set of structures which hold information about various segments of a program. Loadable segments are mapped to virtual memory using the addresses stored in the `p_vaddr` fields of the structures (for more details, see [*30*]). Since executable files typically use (non-relocatable) absolute code, the loadable segments must reside at addresses specified by `p_vaddr` in order to ensure correct execution.

On the other hand, shared object segments contain position-independent code (PIC), which allows them to be mapped to almost any virtual address. However, in our experience, the dynamic linker almost always chooses to map them starting at `p_vaddr`, e.g., this is the case with `libc.so.6` (the Standard C library) on Red Hat Linux distributions. The lowest loadable segment address specified is `0x42000000`. Executables start at virtual address `0x08048000`, which leaves a large amount of space (around 927MB) between the executable code and the space where shared libraries are mapped. Typically, every process which uses the dynamically-linked version of `libc.so.6` will have it mapped to the same base address (`0x42000000`), which makes the entry points of the `libc.so.6` library functions predictable. For example, if we want to know the virtual address where function `system()` is going to be mapped, we can run the following command:

```
$ nm /lib/i686/libc.so.6 | grep system
42049e54 T __libc_system
2105930 T svcerr_systemerr
42049e54 W system
```

The third line of the output shows the virtual address where `system` is mapped.

In order to prevent existing code attacks which jump to library code instead of injected code, the base address of the libraries should be randomized. There are two basic options for doing this, depending on when the randomization occurs. The options are to do the randomization (1) once per process invocation, or (2) statically. The trade-offs involved are as follows:

1. *Dynamically randomize library addresses using* `mmap`. The dynamic linker uses the `mmap` system call to map shared libraries into memory. The dynamic linker can be instrumented to instead call a wrapper function to `mmap`, which first randomizes the load address and then calls the original `mmap`. The advantage of this method is that in every program execution, shared libraries will be mapped to different memory addresses.

2. *Statically randomize library addresses at link-time.* This is done by dynamically linking the executable with a ``dummy'' shared library. The dummy library need not be large enough to fill the virtual address space between the segments of the executable and standard libraries. It can simply introduce a very large random gap (sufficient to offset the base addresses of the standard libraries) between the load-addresses of its `text` and `data` segments. Since shared libraries use relative addressing, the segments are mapped along with the gap. On Linux systems, the link-time gap can be created by using the `ld` options `-Tbss`, `-Tdata` and `-Ttext`. For example, consider a dummy library

which is linked by the following command:

```
$ ld -o libdummy.so -shared
        dummy.o  -Tdata 0x20000000
```

This causes the load address of the text segment of `libdummy.so` to be `0x00000000` and the load address of data segment to be `0x20000000`, creating a gap of size `0x20000000`. Assuming the text segment is mapped at address `0x40014000` (Note: addresses from `40000000` to `40014000` are used by the dynamic linker itself: `/lib/ld-2.2.5.so`), the data segment will be mapped at address `0x60014000`, thereby offsetting the base address of `/lib/i686/libc.so.6`.

The second approach does not provide the advantage of having a freshly randomized base address for each invocation of the program, but does have the benefit that it requires no changes to the loader or rest of the system. We have used this approach in our implementation. With this approach, changing the starting address to a different (random) location requires the library to be re-obfuscated (to change its preferred starting address).

2.3.3 Text/data segment randomization

Relocating a program's text and data segments is desirable in order to prevent attacks which modify a static variable or jump to existing program code. The easiest way to implement this randomization is to convert the program into a shared library containing position-independent code, which, when using `gcc`, requires compiling with the flag `-fPIC`. The final executable is created by introducing a new `main` function which loads the shared library generated from the original program (using `dlopen`) and invokes the original `main`. This allows random relocation of the original program's text and data segments. However, position-independent code is less efficient than its absolute address-dependent counterpart, introducing a modest amount of extra overhead.

An alternative approach is to relocate the program's code and data at link-time. In this case, the code need not be position-independent, so no performance overhead is incurred, Link-time relocation of the starting address of the executable can be accomplished by simple modifications to the scripts used by the linker.

Our implementation supports both of these approaches. Section 4 presents the performance overheads we have observed with each approach.

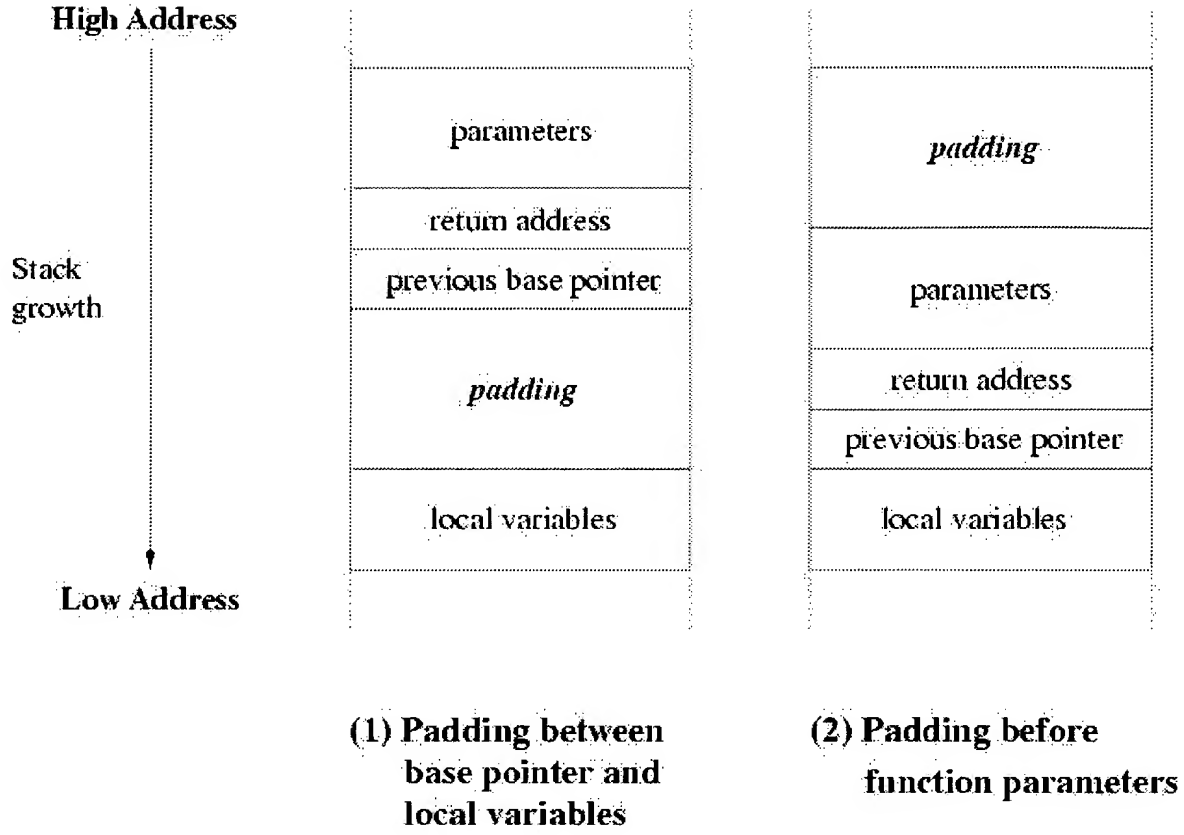

Figure 3: Potential locations of padding inserted between stack frames.

2.3.4 Random stack frame padding

Introducing padding within stack frames requires that extra storage be pushed onto the stack during the initialization phase of each subroutine. There are two basic implementation issues that arise.

The first issue is the randomization of the padding size, which could be static or dynamic. Static randomization introduces practically no runtime overhead. Dynamic randomization requires the generation of a random number at regular intervals. Additionally, the amount of extra code required for each function preamble is significant. Moreover, if the randomization changes the distance between the base of the stack frame and any local variable (from one invocation of a function to the next) then significant changes to the code for accessing local variables are required, imposing even more overheads. For these reasons, we have currently chosen to statically randomize the padding, with a different random value used for each routine.

The second issue concerns the placement of the padding. As shown in Figure 3, there are two basic choices: (1) between the base pointer and local variables, or (2) before parameters to the function:

1. *Between the base pointer and local variables.*
   This requires transformation of the callee to modify the instruction which creates the space for the local variables on the stack. Local variables are accessed using instructions containing fixed constants corresponding to their offset from the base pointer. Given that the padding is determined statically, the transformation simply needs to change the constants in these instructions. The main

benefit of this approach is that it introduces a random gap between local variables of a function and other security-critical data on the stack, such as the frame pointer and return address, and hence makes typical stack-smashing attacks difficult.

2. *Before parameters to the function.*
   This is done by transforming the caller. First, the set of argument-copying instructions is located (usually `PUSH` instructions). Next, padding code is inserted just before these instructions. The primary advantage of this approach is that the amount of padding can change dynamically. Disadvantages of the approach are (a) in the presence of optimization, the argument-pushing instructions may not be contiguous, which makes it difficult to determine where the padding is to be introduced, and (b) it does not make stack-smashing attacks any harder since the distance between the local variables and return address is left unchanged.

We have implemented the first option. As mentioned earlier, extraction of accurate control-flow graphs can be challenging for some routines. To ensure that our transformation does not lead to an erroneous program, the following precautions are taken:

- Transformation is applied to only those routines for which accurate control-flow graphs can be extracted. The amount of padding is randomly chosen, and varies from 0 to 256, depending on the amount of storage consumed by local variables, and the type of instructions used within the function to access local variables (byte- or word-offset). From our experience on instrumentation of different binaries, we have found that around 95-99% of the routines are completely analyzable.

- Only functions which have suitable behavior are instrumented. In particular, the function must have at least one local variable and manipulate the stack in a standard fashion in order to be instrumented. Moreover, the routines should be free of non-standard operations that reference memory using relative addressing with respect to the frame pointer.
- Only *in place* modification of the code is performed. By *in place*, we mean that the memory layout of the routines is not changed. This is done in order to avoid having to relocate the targets of any indirect calls or jumps.

These precautions have limited our approach to instrument only 65% to 80% of the routines. We expect that this figure can be improved to 90+% if we allow modifications that are not in-place, and by using more sophisticated analysis of the routines.

2.3.5 Heap randomization

The base address of the heap can be randomized using a technique similar to the stack base address randomization. Instead of changing the stack pointer, code is added to allocate a randomly-sized large chunk of memory, thereby making heap addresses unpredictable. In order to randomize the relative distances between heap data, a wrapper function is used to intercept calls to `malloc`, and randomly increase the sizes of dynamic memory allocation requests by 0 to 25%. On some OSes, including Linux, the heap follows the data segment of the executable. In this case, randomly relocating the executable causes the heap to also be randomly relocated.